Accelerated Path-Based Timing Analysis with MapReduce

Tsung-Wei Huang
Dept. of Electrical and Computer Engineering
University of Illinois Urbana-Champaign
IL, USA
twh760812@gmail.com

Martin D. F. Wong
Dept. of Electrical and Computer Engineering
University of Illinois Urbana-Champaign
IL, USA
mdfwong@illinois.edu

ABSTRACT

Path-based timing analysis (PBA) is a pivotal step to achieve accurate timing signoff. A core primitive extracts a large set of paths subject to path-specific or less-pessimistic timing update. However, this process in nature demands a very high computational complexity and thus has been a major bottleneck in accelerating timing closure. Therefore, we introduce in this paper a fast and scalable PBA framework with *MapReduce* – a recent programming paradigm invented by Google for big-data processing. Inspired by the spirit of MapReduce, we formulate our problem into tasks that are associated with keys and values and perform massively-parallel map and reduce operations on a distributed system. Experimental results demonstrated that our approach can easily analyze million nodes in a single minute.

Categories and Subject Descriptors

J.6 [**Computer-aided design (CAD)**]: Timing analysis; D.1.3 [**Parallel programming**]: Distributed computing

Keywords

Path-based static timing analysis, MapReduce

1. INTRODUCTION

Static-timing-analysis (STA) is a crucial step in verifying the expected timing behaviors of an integrated circuit [6]. During the STA, both graph-based timing analysis (GBA) and path-based timing analysis (PBA) are used. GBA performs linear scan on the circuit graph and estimates the worst timing quantities at each endpoint. GBA is very fast but the results are pessimistic. Hence, PBA is often performed after GBA to remove unwanted pessimism. Starting from a negative endpoint, a core PBA procedure peels a set of paths in non-increasing order of criticality and applies path-specific timing update to each of these paths [7]. However, path peeling is a computationally expensive process. The high runtime demand severely restrains the capability of PBA during timing signoff.

Unfortunately, current literature still lacks for novel ideas of fast PBA [12]. As pointed out by 2014 TAU timing analysis contest, algorithms featuring multi-threaded or massively-parallel accelerations are eagerly in demand [9]. Howbeit, parallel PBA has been reported as a tough challenge primarily because a path can be prototypically various. For instance, a path can exhibit arbitrary lengths and span different logical cones and physical boundaries. Computations in this way are typically hard to be issued in parallel. Although a few prior works claimed to have a solution, the results are usually compromised with accuracy [5, 11].

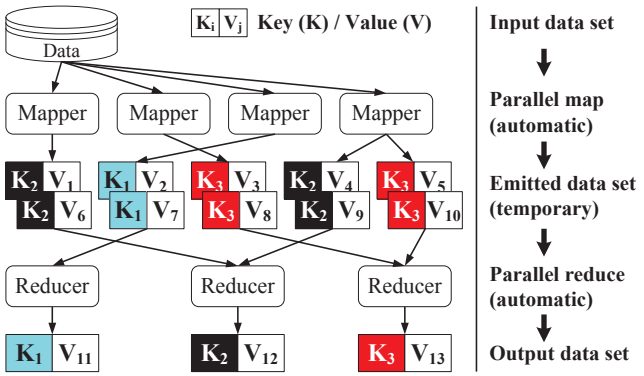

Figure 1: The execution flow of a MapReduce job.

As a consequence, we introduce in this paper an ultra-fast PBA framework with MapReduce. The concept of MapReduce is shown in Figure 1. A MapReduce program applies parallel map operations to input tasks and generates a set of temporary key/value pairs. Then parallel reduce operations are applied to all values that are associated with the same key in order to collate the derived data properly [8]. Users only need to provide desired map/reduce functions while parallelization details are encapsulated in a MapReduce library [1, 2]. This programming paradigm inspires us to rethink the PBA problem as "map" operations followed by "reduces". Specifically, we cast the PBA problem into tasks with keys and values that are sandwiched around massively-parallel map and reduce operations.

Our contributions are summarized as follows. 1) We successfully investigated the applicability of MapReduce to accelerate PBA. Our framework is very general in gaining massively-parallel computations, imposing no physical and

ISPD'15, March 29–April 1, 2015, Monterey, CA, USA.

http://dx.doi.org/10.1145/2717764.2717771.

logic constraints. 2) Our framework increases the productivity as designers can focus on timing-oriented turnaround, leaving all hassle of parallelization details to the MapReduce library. 3) We have seen a substantial speedup from the experimental results. On a large distributed system, millions of cells can be easily analyzed in a few minutes. These features all add up to faster design cycle. Our work can be beneficial for the speedup of the signoff timing closure, on which up to 40% of the design flow are typically spent.

2. PATH-BASED TIMING ANALYSIS

PBA has gained much attention in deep submicron era due to its capability of configuring features such as clock-reconvergence-pessimism removal (CRPR) and advanced-on-chip-variation (AOCV) derating for less-pessimistic timing reports [9, 10]. Since most of these features are path-specific, a core yet computationally expensive building block of PBA is to peel a path set from each endpoint and recompute the timings path-by-path. By analyzing the path with reduced pessimism, many timing violations can be waived which in turn tells better timing signoff. Because of this crucial benefit, studies in accelerating PBA are in demand especially when we move to many-core era. Simply put, the following aspects are in particular of interests.

- Performance is the top concern. A substantial runtime saving will make a breakthrough in timing signoff.
- Modern circuits are complex. Practical parallelization must scale up with the growth of the circuit size.
- The framework needs to be general and flexible, imposing least constraints and complexity.
- Adequate granularity control is necessary in order to effectively organize computations at a massive scale.
- Orthogonality should be featured. Compromised solutions to the design methodology are discouraged.

The above issues all combine to challenges in the development of parallel PBA algorithms. If the PBA runtime can be significantly improved, designers are able to utilize PBA on a larger set of paths and perform their analyses earlier in the design closure flow. As a result, researchers must continue to provide viable parallel solutions along with the rapid evolution of the computational power.

3. PROBLEM FORMULATION

The circuit network is input as a directed-acyclic graph $G = \{V, E\}$, where V is the pin set of circuit elements and E is the edge set specifying pin-to-pin connections. Each edge e is associated with a tuple of earliest and latest delays. A path is an ordered sequence of nodes or edges and the path delay is the sum of delays through all edges. In this paper, we are in particular emphasizing on the *data path*, which is defined as a path from either the primary input pin or the clock pin of a launching flip-flop (FF) to the data pin of a capturing FF. A test is defined w.r.t. an FF as hold or setup check on any data paths captured by this FF. Considering a test set T as well as a positive integer k, the following two tasks are essential for PBA [7, 9].

Task 1 – Sweep report: *The program is asked to sweep all tests and output the top k critical paths for each test.*

Task 2 – Block report: *The program is asked to report the top k critical paths across all tests.*

4. MAPREDUCE FRAMEWORK

In this section we discuss our PBA framework with MapReduce. We first brief the MapReduce programming paradigm and then detail each step of our framework.

4.1 MapReduce Programming Paradigm

Since being first introduced by Google in 2004, the MapReduce programming paradigm has been widely applied to many domains such as data mining, database system, and high-performance computing [8]. The spirit of a MapReduce program lies in "*keys*" and "*values*" which are generated and manipulated by user-defined functions "*mapper*" and "*reducer*". A key and a value are simply bytes of strings of arbitrary length which are logically associated with each other and thus can represent generic data types. The MapReduce library automatically schedules parallel map and reduce operations linking mapper and reducer to handle the input data on a distributed system. State-of-the-art libraries for this purpose such as Apache Hadoop and MR-MPI from Sandia National Lab. are readily available [1, 2].

Algorithm 1: CanonicalForm(D, mapper, reducer)

Input: input data D, user-defined mapper and reducer

1 $\{M \mid <tmp\_key : tmp\_value>\} \leftarrow$ Map(D, mapper) ;
2 $\{C \mid <unique\_key : value\_list>\} \leftarrow$ Collate(M);
3 $\{R \mid <key : value>\} \leftarrow$ Reduce(C, reducer);
4 **return** R

A canonical MapReduce program is presented in Algorithm 1. The first is the map step, which takes a set of data and converts it into another set of data produced by the function mapper, where individual elements are represented as temporary key/value pairs. The collate step aggregates across temporary key/value pairs where each unique key appears exactly once and the corresponding value is a concatenated list of all the values associated with the same key [1]. The reduce step then takes a single entry from the aggregated key/value pairs and creates a new key/value pair which stores the output generated by the function reducer. Parallelism is evident since function calls by map and reduce are independent to each other and can be executed on different processors simultaneously. In general, map and reduce are intra-process operations while collate involves inter-process communication because of aggregation.

4.2 Formulation of Task Graphs

In order to develop a MapReduce program, computations that can be issued to parallel map and reduce operations must be exploited from our problem. Considering a test t, we observe: 1) every data path captured by this testing FF reaches the same endpoint; 2) the *source pins* from where a path originates is prototypically consistent, being either the primary input pins or the clock pin of a launching FF. The first feature implies that paths feeding the same endpoint belong to the same test. By tagging each path with a key indicating the corresponding test index, the program can keep track of the test to which a path belongs. The second

[1] In some articles the collate is absorbed into the reduce step.

feature implies that paths are wrapped in a multi-source single-target graph. This motivates us to decompose a test into several task graphs with regard to different and smaller groups of source pins.

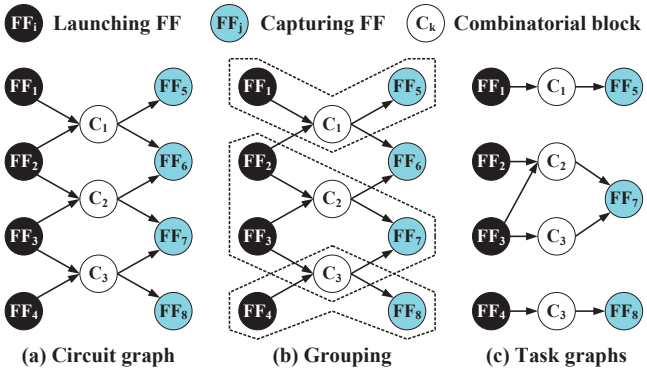

Figure 2: An example formulation of the task graphs.

We define g_t for each test t as a set of task graphs $g_t = \{g_t^1, g_t^2, ..., g_t^i\}$ and G_T as a union set of all task graphs. Deriving from a test t, a task graph g_t^i is a subgraph spanning all connectivities from a subset of source pins to the data pin of this test. Under the same test, the source pins corresponding to different task graph are mutually disjointed. We associate each task graph with a key indicating the test index to which this task graph belongs. An example is illustrated in Figure 2. We can see three task graphs are derived from the tests on capturing FFs 5, 7, and 8, respectively. Notice that a task graph is indeed a portion of the original circuit graph. Every edge of the task graph comes with the same delay values as the original circuit graph.

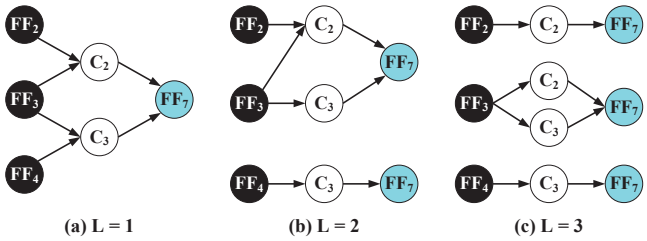

Figure 3: Granularity control of the task graph.

The granularity control of the task graphs is an important factor as it arises performance concern such as process communication and computation load. We define L-way partition as a partition of each test into L task graphs such that each task graph has roughly even size on the corresponding set of source pins. Figure 3 shows an example of 1-way, 2-way, and 3-way partitions of the test on FF 7 from Figure 2. While discovering a suitable granularity level tends to be case-dependent, we consider in this paper only the case where the number of tests is less than the number of available computing cores. Assuming P cores are available in such the case, up to P task graphs are generated from each test in order to balance the computation load. We should be mindful that dividing a test into multiple task graphs facilitates the parallelism but also gives rise to process communication because of data merging afterwards.

The generation of task graphs is presented in Algorithm 2. We first identify all source pins of a given test t through a backtrace starting at the data pin d of this test (line 1:2). The number of task graphs being generated is determined by a comparison between the number of input tests and the number of available computing cores (line 3:8). Then we iteratively group a set of source pins S_d^k in accordance to the specified number of task graphs and perform depth-first-search (DFS) to induce the corresponding task graph (line 9:15). Each induced task graph is assigned a key indicating its test index and is emitted as a key/value object in the end of each iteration (line 14).

Algorithm 2: Generator(t)

Input: a test t
Output: a set of task graphs $g_t = \{g_t^1, g_t^2, ..., g_t^i\}$

```
1  d ← data pin of the test t;
2  S_d ← source pins obtained through a back traversal at d;
3  P ← number of available computing cores;
4  L ← 1;
5  if |T| < P then
6  |  L = P;
7  end
8  num_src ← ⌈|S_d|/L⌉;
9  for i ← 1 to L do
10 |  S_d^i ← {num_src frontmost elements in S_d};
11 |  S_d ← S_d \ S_d^i;
12 |  g_t^i ← subgraph induced from S_d^i to d;
13 |  key[g_t^i] ← t;
14 |  Emit make_pair(t, g_t^i);
15 end
```

Based on the knowledge constructed so far, we deliver a high-level sketch of our MapReduce-based PBA framework. The map operation is responsible for 1) the generation of task graphs from each test and 2) the path extraction from each task graph. Because of the granularity control, a test might be broken into several task graphs that are distributed to different processors during the map operations. Collate method is required in order to reorganize paths to their right places. Eventually, the reduce operation peels out a desired path set and emits it as the final solution. We conclude this section by the following lemma.

LEMMA 1. *Every path exactly and uniquely exists in one task graph.*

4.3 Mapper and Reducer Functions

Based on the definition of task graphs, we develop the function calls for map and reduce operations. As presented in Algorithm 3, our mapper function takes an arbitrary task graph and extracts the top-k critical paths (line 1). We leave this extraction process as a black box for user preferences. In this paper, the optimal path ranking algorithm by [10] is used as our default engine. Then it iterates through each path and performs path-specific update according to user-configured features such as CRPR and AOCV (line 3). Each iteration ends with an emission of a key/value pair where the value is a path string and the key is being either 1) the key of the input task graph if sweep is the task objective (line 5:6) or 2) a nominal number instead (line 7:9).

Any key/value pair emitted by our mapper is in fact a solution fragment, where the key indicates the test index to which the value of a path string belongs. It can be inferred

Algorithm 3: Extracter(g_t^i)

Input: an arbitrary task graph g_t^i
Output: an emitted set of key/value pairs

```
1  P ← top k critical paths extracted from g_t^i;
2  foreach path p_i ∈ P do
3      p'_i ← update p_i according to user-configured features;
4      value ← make_string(p'_i);
5      if sweep is the task objective then
6          key ← key[g_t^i];
7      else
8          key ← −1;
9      end
10     Emit make_pair(key, value);
11 end
```

that after calling the collate method, there are two possible outcomes: either paths that belongs to the same task graph are aggregated together or all paths are put in a single group, depending on the task objective. Eventually, our reducer takes each unique key/value pair and peels out the final top-k critical paths from the path set stored in each value list. This implementation is given in Algorithm 4.

Algorithm 4: Peeler(r)

Input: an unique key/value pair r
Output: an emitted key/value pair

```
1  key ← r.key;
2  P ← paths parsed from r.value;
3  sort P in non-increasing order of criticality;
4  P' ← {k frontmost elements in P};
5  value ← make_string(P');
6  Emit make_pair(key, value);
```

LEMMA 2. *There are either $|T|$ or $O(P|T|)$ mapper calls on a distributed cluster with P computing processors.*

PROOF. The execution of each benchmark has two possible conditions, either the number of tests is greater the number of computing processors or the number of tests is less than the computing resources. For the former case, each test is processed by an independent mapper function and thus there are totally $|T|$ mapper calls. For the later case where the number of tests is less than the available core count, each test is decomposed into $O(P)$ task graphs. Hence, there are totally $O(P|T|)$ mapper calls. □

LEMMA 3. *There is only one reducer call for block report while there are $|T|$ reducer calls for sweep report.*

PROOF. For block report, the key/value pairs emitted by the extractor all have the same key value (i.e., -1). Therefore, the collate operation produces only one key/value pair for the following reduce operation. On the other hand, the intermediate key values for sweep report adhere to the test indices of the task graphs. Therefore, the collate operation produces a total of $|T|$ distinct key/value pairs for the following reduce operation. □

4.4 Main Program

The main program of our PBA framework is shown in Algorithm 5. The first two lines perform map operations that call Algorithm 2 to generate a set of task graphs. Using the task graphs as input, the next three lines follow the canonical form of a MapReduce program, where map operations call Algorithm 3 to perform path extraction on each task graph, and reduce operations call Algorithm 4 to peel out the final solution. Prior to the function return, paths are parsed from the output values of our reducer (line 6:15). Each path is conventionally tagged with the corresponding test index which can be retrieved from the key value (line 10).

Algorithm 5: MapReducePBA(G)

Input: a circuit graph G, a test set T
Output: an analyzed path set

```
1  D ← Map(T, Generator) ;
2  G_T ← task graphs parsed/read from D;
3  M ← Map(G_T, Extracter) ;
4  C ← Collate(M);
5  R ← Reduce(C, Peeler);
6  if sweep is the task objective then
7      P ← φ;
8      foreach pair r in R do
9          P_r ← paths parsed from r.value;
10         Tag P_r with the test index t retrieved from r.key;
11         P ← P ∪ P_r;
12     end
13     return P
14 end
15 P ← paths parsed from the value in R;
16 return P
```

THEOREM 1. *The proposed framework is correct.*

PROOF. Lemma 1 has shown the exactness and uniqueness of every path. Proving the correctness of our framework is equivalent to showing that the path set from the input of a reducer contains the top-k critical paths for the corresponding test. Recalling that the input of our reducer is an unique key/value pair. The key indicates the test index and the value is a concatenated list of values with each value storing the top-k critical paths of a task graph generated from this test. It is obvious by set properties that the top-k critical paths for this test must be a subset of the path set stored in the value list. Since our reducer is in fact a sorting process, the output is the value that stores the final top-k critical paths for this test. Notice that for block report the test index is nominal while this fact has no impact on the truth of this proof. □

5. DATA MANAGEMENT

Efficient data management is crucial to a MapReduce program. We discuss in this section some technical details and data management through our implementations.

5.1 Data Locality

Exploiting the data locality is an important principle to efficient MapReduce programs. Improving the data locality can reduce the network overhead during the execution, which in turn tells better runtime performance. In order to improve the data locality, each processor stores a replicate of the circuit graph in its own local memory. Despite higher memory demand, accesses to the circuit graph such as generation of task graphs and extraction of critical paths are reached in hand without extra data passing which is normally time-consuming.

5.2 Storage Efficiency

The communication load is a non-negligible cost for a MapReduce program in particular during the collate operation. Passing long values of paths gives rise to the problem of frequent memory allocation which is typically time consuming. In order to minimize the communication load, explicit path traces are stored in the memory of each individual machine. Each path is tagged with an unique index which is used to represent the storage address and machine number or the temporary file name. Paths are passing through these indices during collate operation and the final recovery of path traces is done by indexing back to these tags.

5.3 Hidden Reduce

Another way to alleviate the communication overhead is to avoid unnecessary data passing during the collate operation. Within a same processor, a reduce operation before the collate call is pre-applied to those path sets having the same key label. We term this reduce operation as "hidden reduce" because it is implicitly processed after each mapper call of path extraction. In other words, multiple data with the same key label in a each processor are merged first so as to reduce the amount of data passing. It is obvious by Theorem 1 the optimality of the final solution is not affected by this hidden reduce operation.

6. EXPERIMENTAL RESULTS

Our program is implemented in C++ language on a 64-bit linux operating system. The C++ based MR-MPI API is used as our MapReduce library [2]. Evaluation is taken on an academic computer cluster which has over 500 compute nodes. Each compute node is configured with 16 Intel 2.60GHz cores and 128GB RAM. The network infrastructure uses 384-port Mellanox MSX6518-NR FDR InfiniBand in order to offer high speed interconnect between clusters. Access to the compute nodes for running a program is done via a script submission specifying the number of process cores or threads to be used.

Table 1: Statistics of the benchmarks from 2014 TAU timing analysis contest [9].

Circuit	$\|V\|$	$\|E\|$	$\|I\|$	$\|O\|$	# Tests	# Paths
combo5	2051804	2228611	432	164	79050	19227963
combo6	3577926	3843033	486	174	128266	19227963
combo7	2817561	3011233	459	148	109568	19227963

$|V|$: # of pins. $|E|$: # of edges. $|I|$: # of primary inputs. $|O|$: # of primary outputs. # Tests: # of setup/hold tests. # Paths: maximum # of data paths per test.

Experiments are undertaken on the three largest benchmarks, combo5, combo6, and combo7 from 2014 TAU timing analysis contest [9]. Each of the three testcases is created by combining a set of industrial circuits (e.g., vga_lcd, systemcde2, aes_core, des_perf, usb_funct, wb_dmav, systemcaes, and tv80) that were already open-source to academia. combo5 is the combination of circuits vga_lcd, usb_funct, des_perf, tv80, wb_dmav, and systemcaes. combo6 is the combination of circuits vga_lcd, aes_core, des_perf, usb_funct, systemcde2, and tv80. combo7 is the combination of circuits vga_lcd, tv80, aes_core, systemcaes, and vga_lcd. Statistics of these testcases are summarized in Table 1. All testcases are million-scale circuit graphs and the number of tests could reach up to 128266 in combo6.

6.1 Baseline Setting

We configure CRPR as the baseline application in our PBA framework. CRPR is an important step during the signoff timing cycle. Without CRPR, signoff timing analyzer reports worse violation than the true timing properties owned by the physical circuits. The 2014 TAU timing analysis contest has addressed this issue in order to motivate novel ideas for fast and accurate path-based CRPR [9]. The optimal path ranking algorithm proposed by the first-place winner, UI-Timer, is applied to our path extractor [10]. In order to enable CRPR, the third line of Algorithm 3 is implemented as follows: For each path being iterated, the common clock segment is found by a simple walk through the corresponding launching clock path and the capturing clock path. The path slack is then adjusted by the amount of pessimism on the common segment.

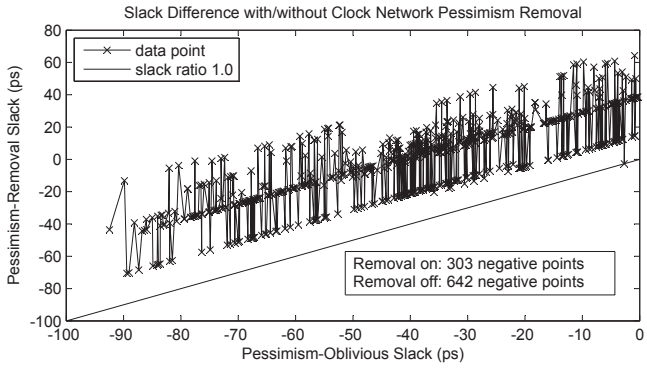

Figure 4: Impact of CRPR on path slacks.

Figure 4 illustrates the impact of CRPR on path slacks of the hold test from a subcircuit block of the testcase combo6. It can be observed that the values of path slacks are in general increased after the clock network pessimism was removed. The number of failing tests was able to be reduced from 642 to less than half [10]. Another evidence which can be discovered from Figure 4 is the path-specific property of the clock network pessimism. The most critical path prior to CRPR is not necessarily reflective of the true counterpart after CRPR. Such a fact reveals the necessity of PBA in order to peel out the true critical path. More knowledge about CRPR can be referred to [9].

6.2 Performance Characterization

We begin by discussing the generic performance of our MapReduce-based PBA. Evaluation is undertaken through cross combinations of path count (i.e., k) and core count in running our program. We request 1 to 10 compute nodes with each configured by 10 cores. That is, the core count varies from 10 to 100 using 10 as the scaling interval. A special case with only 1 core is also evaluated in order to demonstrate the baseline without any parallelism. The path count starts at 1 and varies from 10 to 100 using 10 as the scaling interval. A total of 121 combinations of path count and core counts are executed for each benchmark.

The number of key/value pairs processed on each circuit benchmark is illustrated in Figure 5. It can be observed that for each circuit graph the number of key/value pairs processed by map and reduce operations grows as the path count increases. Notice that the path count is the only factor

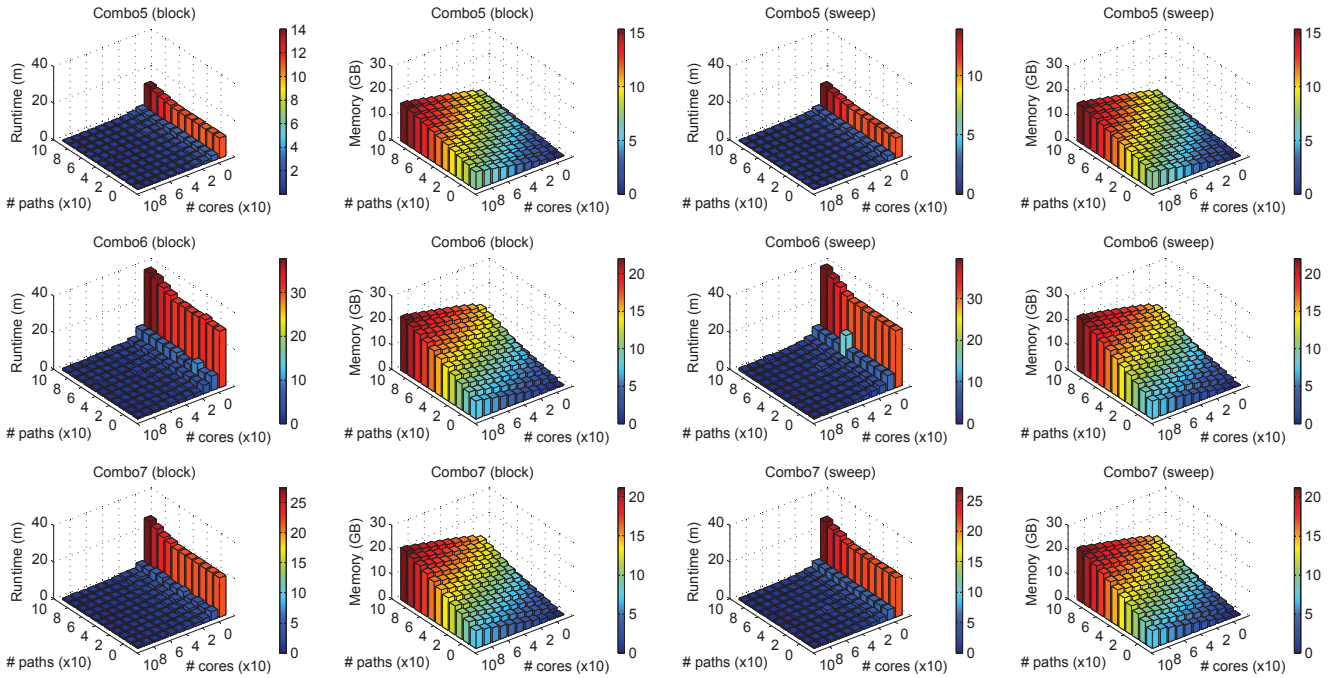

Figure 6: Performance characterization of our MapReduce-based PBA on circuit benchmarks combo5, combo6, and combo7 under block report and sweep report. Within a single minute, all tests can be accomplished using approximately 40 cores, 100 cores, and 80 cores, for combo5, combo6, and combo7, respectively.

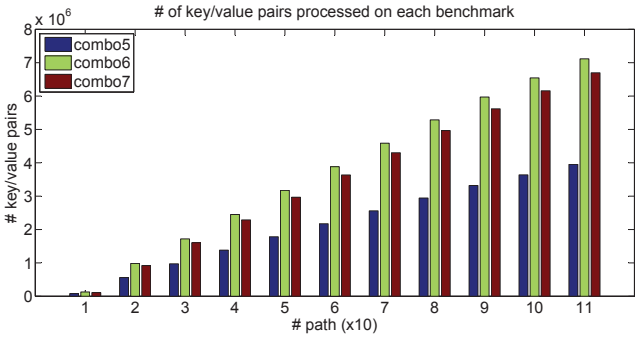

Figure 5: Bar chart of the number of key/value pairs processed on each circuit benchmark.

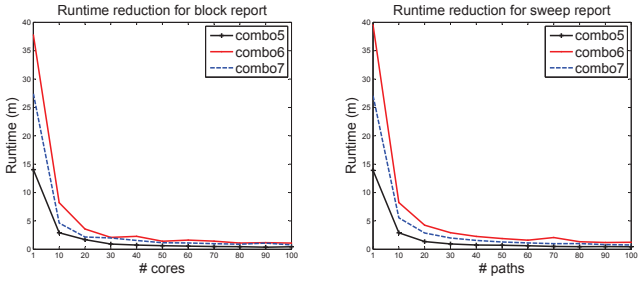

Figure 7: Runtime reduction versus core count.

that contributes to the growth of the number of key/value pairs since the construction of key/value pairs is dedicated to paths. The largest number appears in the report of 100 paths path, in which the program generated 3953344 key/value pairs for combo5, 7114972 key/value pairs for combo6, and 6696880 key/value pairs for combo7. In general, the more the number of key/value pairs is, the higher the runtime and memory storage the program demands.

The overall performance of our MapReduce-based PBA is shown in Figure 6. The left two columns of plots show the runtime value and memory usage of our program under block report, while the right two columns show the plots under sweep report. We first discuss the runtime performance of our program. In a rough view, the runtime scales down drastically as the core count increases. Using only a single core without any parallelism, the program took up to (i.e., among all path settings) 14.03 (13.92) minutes, 37.76 (39.53) minutes, and 27.41 (27.07) minutes to accomplish block (sweep) reports for combo5, combo6, and combo7, respectively. It can be seen that the runtime significantly goes down when MapReduce begins distributing works across processors. Even using only 10 processors, the runtime values can be significantly reduced to 2.92 (2.91) minutes, 8.22 (8.24) minutes, and 4.63 (5.55) minutes under block (sweep) reports of combo5, combo6, and combo7, respectively. The slope of the runtime reduction can be clearly seen in the sliced 2D plot fixing path count to 100 in Figure 7. Within a single minute, all tests can be accomplished using approximately 40 cores, 100 cores, and 80 cores, for combo5, combo6, and combo7, respectively.

Figure 8 discovers the runtime portions taken by map operations, collate operations (i.e., process communication or "Comm" for short), and reduce operations. We measure the runtime portion as an average value across all different settings of path counts and core counts. We have observed that

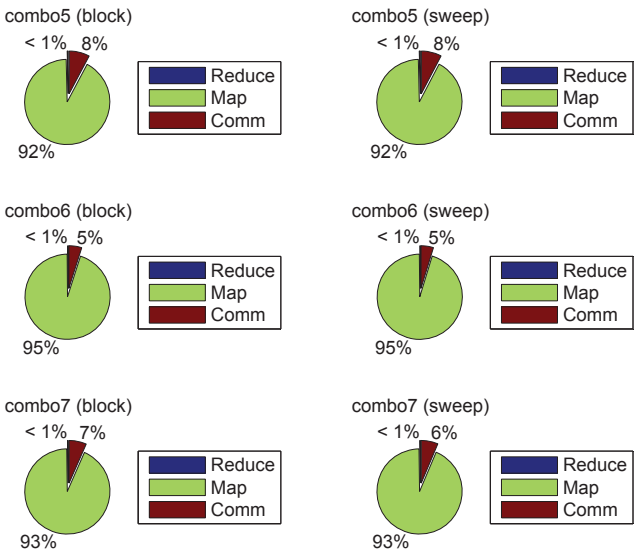

Figure 8: Runtime portion of map operations, reduce operations, and process communication.

reduce operations spend the least amount of time (< 1%) comparing to the others since it involves only string parsing and value sorting. On the other hand, the time spent on map operations occupies the majority of the entire runtime. This is because map operations are responsible for the generation of task graphs and the extraction of critical paths, which are relatively expensive computations. For all benchmarks, more than 90% of the entire runtime is taken by map operations. The rest portion of the runtime is occupied by the collate operation, from which we can see about 4–5% of the entire runtime is spent on the process communication. In fact, without applying the trick mentioned in Section 5.2, the process communication burdens the entire runtime by over 20%.

Next we discuss the memory cost of our program. The amount of memory usage is measured by the peak moment during the execution across all processors (i.e., including the master processor). Generally speaking, the amount of memory usage grows as the increase of either path count or core count, which can be seen in Figure 6. The peak memory usage we observed are approximately 15GB, 22GB, and 21GB for combo5, combo6, and combo7, all under sweep report with 100 cores and 100 paths, respectively. We provide two extra sliced plots from the sweep report in Figure 9 to show clearer memory cost in terms of the growth of 1) core count with path count fixing to 100 and 2) path count with core count fixing to 100. As the path count or the core count increases, the amount of memory usage grows gradually except for the sharp spot at 10-core level where the distributed MapReduce begins taking effect.

To sum up, the experimental results have demonstrated the performance of our PBA framework with MapReduce. It is highly scalable as we have seen a significant runtime reduction as the core count grows. Even in the first level at which only 10 cores are involved in parallelism, the runtime is decreased by 75–86% across all runs. From the storage point of view, the memory consumption of our approach is fairly reasonable. At the highest peak we have observed in running combo6 with 100 cores and 100 paths, the total amount of memory demanded by our program is about 22GB. In other words, the average amount of memory usage per processor is less than 1GB. These evidences have justified the practical viability of our approach. The substantial speedup we have obtained is beneficial for the discovery of a way to fast timing closure.

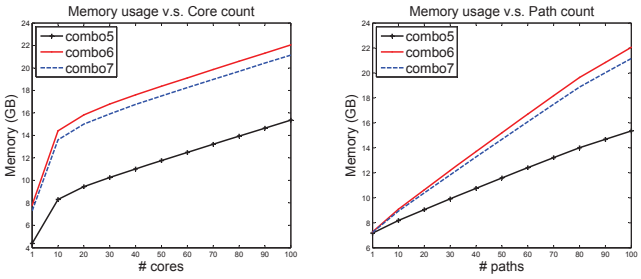

Figure 9: Memory usage in terms of path count and core count.

6.3 Competence over Multi-Threading

We evaluated in this section the competence of our approach over the implementation using multi-threading, another popular type of parallel programming with shared-memory model. The inherent architecture of a multi-threaded program is distinct from that of distributed computation such as the MapReduce programming environment we discussed in this paper. In multi-threaded programming, multiple threads or processors can operate independently on a stand-alone machine but share the same memory resources. The memory bandwidth of the machine typically dominates the entire runtime performance. As a result, the scalability of multi-threaded computation is typically not as decent as the one of distributed computation. Several libraries for using shared memory such as OpenMP and POSIX are reachable in the public domain [3, 4].

We refit our MapReduce program to the multi-threaded version by replacing the mapper calls and reducer calls with parallel for loop (e.g., #pragma omp statement) using the API from OpenMP 3.0 [3]. In our cluster each compute node is configured with 16 Intel 2.60GHz cores and 128GB RAM in a stand-alone machine. Up to 16 threads or 16 processors can be concurrently executed using either multi-threaded computation or distributed MapReduce operations. Due to the architectural limitation of multi-threading, evaluations are undertaken in a single compute node using different core counts from 1 to 16. The performance differences between multi-threading and MapReduce are interpreted in terms of runtime values and memory usage, as illustrated in Figure 10. For page efficiency, we discuss only the experiment of block report with the single-most critical path.

The competence of MapReduce over multi-threading is clearly demonstrated by the runtime plot in Figure 10. In comparison to multi-threading, our MapReduce program obtains higher runtime speedup (i.e., over multi-threading) and better scalability as core count grows up. The largest difference we observed was in combo6 with 2 cores, where our MapReduce program accomplished all tests by 32 minutes faster than the multi-threaded implementation. Similar trends can also be discovered in other two cases. The reason for having our MapReduce program perform worse at the level of 1 core comes from the redundant overhead of

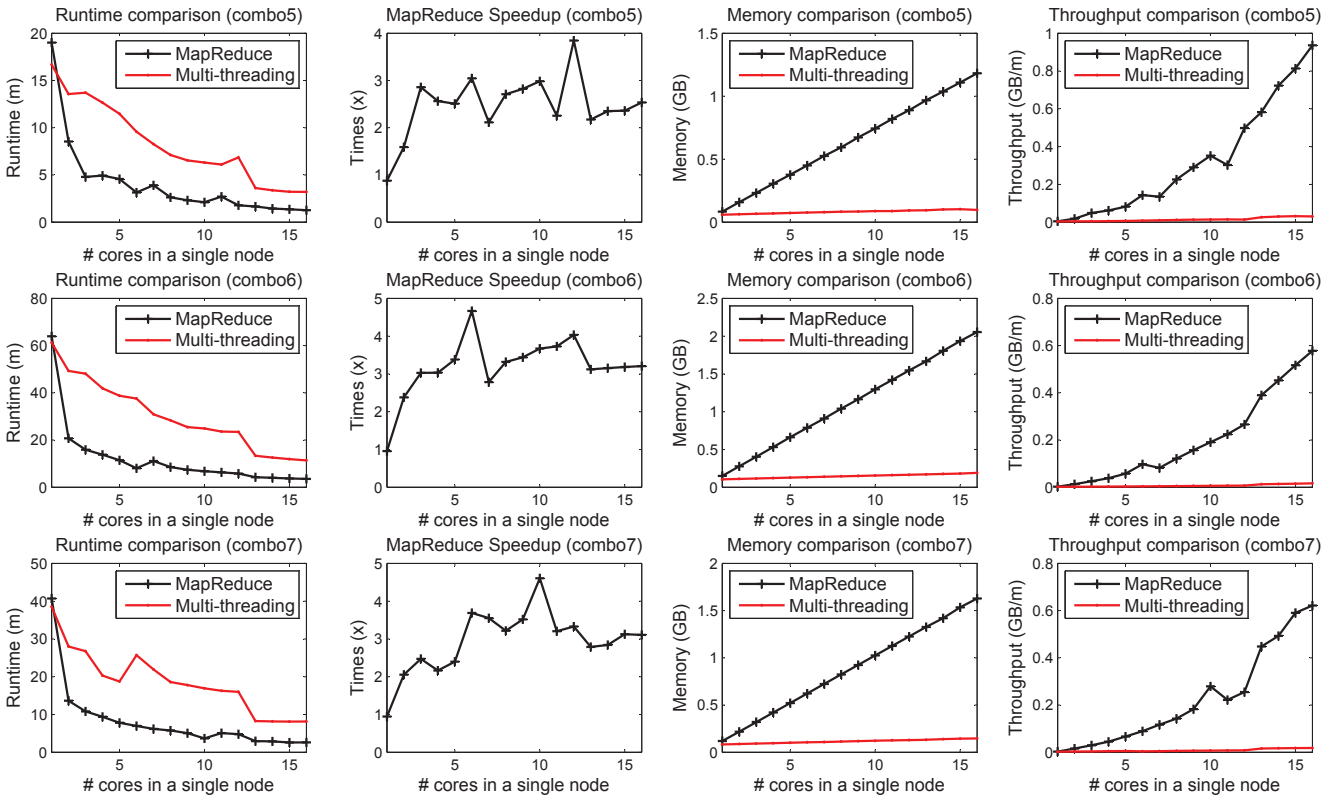

Figure 10: Performance comparison between MapReduce and multi-threading.

key/value processing because of the null parallelism. Nevertheless, such negative margins are solely less than 3 minutes.

It is expected that our MapReduce program consumes higher memory requirements than the multi-threaded implementation. The distributed computation of MapReduce requires an individual block of memory to be allocated for each processor. As shown in the memory comparison in Figure 10, the memory cost of our MapReduce program is linearly proportional to the growth rate of the core count. On the other hand, the amount of memory usage in multi-threading is relatively constant regardless of the increase of core count. Despite less memory cost by multi-threading, the performance of concurrent access to the same global memory block is limited by the memory bandwidth. It can be clearly seen in Figure 10 the process throughput grows poorly compared to the curve achieved by distributed MapReduce. As a consequence, the runtime performance of multi-threading is not as promising as distributed MapReduce even in a standalone machine.

7. ACKNOWLEDGEMENT

This work was partially supported by the National Science Foundation under Grant CCF-1320585.

8. CONCLUSION

In this paper we have presented a fast PBA framework with MapReduce. We have achieved a success in accelerating PBA by a substantial order of magnitude in comparison to non-MapReduce implementations such as single core and multi-threading. The experimental results have demonstrated the pronounced performance of our approach whereby million-scale circuit graphs can be quickly and correctly analyzed within a few minutes on a distributed computer cluster.

9. REFERENCES

[1] Apache Hadoop: http://hadoop.apache.org/
[2] MapReduce MPI Library: http://mapreduce.sandia.gov/
[3] OpenMP: http://openmp.org/wp/
[4] POSIX: https://computing.llnl.gov/
[5] S. Bhardwaj, K. Rahmat, and K. Kucukcakar, "Clock-Reconvergence Pessimism Removal in Hierarchical Static Timing Analysis," US patent 8434040, 2013.
[6] J. Bhasker and R. Chadha, "Static Timing Analysis for Nanometer Designs: A Practical Approach," *Springer*, 2009.
[7] S. Cristian, N. H. Rachid, and R. Khalid, "Efficient exhaustive path-based static timing analysis using a fast estimation technique," US patent 8079004, 2009
[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *CACM*, vol. 51, no. 1, 107–113, 2008
[9] J. Hu, D. Sinha, and I. Keller, "TAU 2014 Contest on Removing Common Path Pessimism during Timing Analysis," *Proc. ACM ISPD*, pp. 153–160, 2014.
[10] T.-W. Huang, P.-C. Wu, and M. D. F. Wong, "UI-Timer: An Ultra-Fast Clock Network Pessimism Removal Algorithm," *Proc. IEEE/ACM ICCAD*, 2014.
[11] O. Levitsky, "Sign Off Quality Hierarchical Timing Constraints: Wishful Thinking or Reality?" *TAU workshop*, 2014.
[12] R. Molina, "EDA Vendors should Improve the Runtime Performance of Path-Based Timing Analysis," *Electronic Design*, 2013